units can be at other levels, such as assembly code. Moreover, executable program unit(s) 210 that are output from linker unit 184 (which is described in more detail below) can be executed in a multi-processor parallel computing environment.

[0028] In an embodiment, translation unit 180 performs a source-to-source code level transformation of program unit(s) 202 to generate translated program unit(s) 204. However, embodiments of the present invention are not so limited. For example, in another embodiment, translation unit 180 could perform a source-to-assembly code level or source-to-object code level transformation of program unit(s) 202.

[0029] Compiler unit 182 receives translated program units 204 and generates object code 208. Compiler unit 182 can be different compilers for different operating systems and/or different hardware. In an embodiment, the compilation of translated program unit(s) 204 is based on the OpenMP standard.

[0030] Linker unit 184 receives object code 208 and runtime library 206 and generates executable code 210. Runtime library 206 can include one to a number of different functions or routines that are incorporated into the object code 208. In one embodiment, executable code 210 that is output from linker unit 184 can be executed in a multi-processor parallel computing environment. Additionally, executable program unit(s) 210 can be executed across a number of different operating system platforms, including, but not limited to, different versions of UNIX, Microsoft Windows ™, real time operating systems such as VxWorks ™, etc.

[0031] Figure 3 is a flow chart for translating an atomic operation according to one embodiment of the invention. An atomic operation protects against races when updating a shared memory location. At block 301, the translation unit 180 receives a program unit(s). At block 303, the translation unit 180 encounters an atomic operation.

At block 307, the translation unit 180 determines if the atomic operation is supported by a low-level instruction. If the atomic operation is supported by a low-level instruction(s), then at block 309 the translation unit 180 translates the atomic operation into an implementation of the low-level instruction(s). For example, the low-level instruction(s) may be implemented as a routine in the runtime library 206.

[0032] The reference at block 307 to determining if an operation is supported by a low-level instruction refers to determining if a low-level instruction(s) can perform the operator on the data-type(s) and data size(s) corresponding to the atomic operation. For example, assume the operation is an addition of two 4-byte integers. This operation is supported if a low-level instruction(s) (e.g., a processor instruction(s) in assembly code) that can atomically perform an addition of 4-byte integers (for example, a 4-byte fetch-and-add instruction) resides on the system 100.

[0033] If the atomic operation is not supported by a low-level instruction(s), then at block 311 the translation unit 180 determines if the size of the data-type being updated by the atomic operation is supported by a low-level instruction(s) that ensures atomicity. If the size of the data-type is supported by such an instruction(s), then at block 313, the translation unit 180 generates a callback routine that encapsulates the atomic operation. At block 314, the translation unit 180 translates the atomic operation by replacing it with a call to an atomic update routine in runtime library 206. The call to the atomic update routine has the address of the callback routine described at block 314 as one of the atomic update routine's arguments. The atomic update routine causes the memory update operation to be performed atomically by enclosing a call to the callback routine with an implementation of the low-level instruction(s) that ensures atomicity. Such instructions can include a compare-and-swap (CAS) instruction, a test-and-set (TAS) instruction, etc.

[0034] If the translation unit 180 determines at block 311 that the size of the data-type corresponding to the atomic operation is not supported by such a low-level instruction(s), then the translation unit 180 surrounds the atomic operation with calls to lock routines (e.g., a call to a lock acquisition routine and a call to a lock release routine) in the runtime library 206.

[0035] In one embodiment of the invention, the translation unit 180 replaces the atomic operation with a call to a routine or function. In various embodiments of the invention, the translation unit 180 inlines instructions instead of inserting a call to a routine, generates assembly code instead of source code, etc.

[0036] In an alternative embodiment of the invention, the compiler unit 182 generates intermediate code for the atomic operation.

[0037] Figure 4 is a diagram illustrating a program unit being translated into a second program unit according to one embodiment of the invention. Figure 4 will be described with reference to Figure 3. In Figure 4, a program unit 401 includes an atomic operation. Program units 403, 405, and 407 are examples of possible translations of the program unit 401. An example of the atomic operation illustrated in the program unit 401 can have the following form:

shared-value = memory-update-operation(shared-value, update-amount);

[0038] The program unit 403 is an example translation of the program unit 401 in accordance with block 309 in Figure 3. The program unit 403 includes a call to a routine that implements a low-level instruction(s) that supports the atomic operation indicated in the program unit 401. For example, if the update-amount in the exemplary atomic operation shown above is "1", then the ROUTINE_FOR_OPERATION may take the form of an atomic fetch-and-add of the shared-value by 1. Another example form of the